NASA Technical Paper 1528

Concurrent Error-Detecting Codes for Arithmetic Processors

Raymond S. Lim

AUGUST 1979

NASA

NASA Technical Paper 1528

Concurrent Error-Detecting Codes for Arithmetic Processors

Raymond S. Lim
Ames Research Center
Moffett Field, California

National Aeronautics
and Space Administration

**Scientific and Technical
Information Branch**

1979

SYMBOLS

A	an n-bit binary number
B	an n-bit binary number
FLOPS	floating-point operations per second
M	numerical range of N
m	modulus of a congruence or the check-base of a residue code
mod	modulo
N	an n-bit binary number consisting of a sign bit and n - 1 magnitude bits
NASF	Numerical Aerodynamic Simulation Facility
n	a positive integer used to indicate the number of bits in N
$\|N\|$	absolute value of N
$\|N\|_m$	residue of N modulo m
PROD	product of A and B
r	residue of N, or remainder
t	a positive integer for partitioning N into t bytes of length ℓ

CONCURRENT ERROR-DETECTING CODES FOR ARITHMETIC PROCESSORS

Raymond S. Lim

Ames Research Center

I. SUMMARY

This paper describes a method of concurrent error detection for arithmetic processors. Low-cost residue codes with check-length ℓ and check-base $m = 2^{\ell} - 1$ are described for checking arithmetic operations of +, -, ×, ÷, complement, shift, and rotate. Of the three number representations, the signed-magnitude representation is preferred for residue checking. Two methods of residue generation are described: the standard method of using modulo m adders and the method of using a self-testing residue tree. A simple single-bit parity-check code is described for checking the logical operations of XOR, OR, and AND, and also the arithmetic operations of complement, shift, and rotate. For checking complement, shift, and rotate, the single-bit parity-check code is simpler to implement than the residue codes.

II. INTRODUCTION

The primary motivation for writing this paper is the NASF Project (refs. 1-5), albeit there are other reasons. Two other reasons are: (1) the Phoenix Project (ref. 6), and (2) the result of a consultation with Professor A. Avizienis of UCLA (ref. 7). The method of concurrent error detection described in this paper is general in theory and is applicable to all arithmetic processors, large and small.

The goal of the NASF Project is to develop a very large computer system capable of operating in the 10^9 FLOPS range for scientific computations. The system is expected to contain approximately one-quarter million high-speed LSI IC's. This order of magnitude of hardware complexity is in the same general class of current very large computer systems, such as the Burroughs BSP, the CYBER 203, the CRAY 1, and the TI ASC, except that the NASF system is planned to have better than an order of magnitude in performance. With this large hardware complexity in the NASF system, the issues of system reliability and trustworthiness are certainly very high in the project goal. In this paper, the justification for concurrent error detection and fault-tolerant computing will not be reiterated since it has been very well described in a previous paper (ref. 8). The basic theory of concurrent error detection for arithmetic processor is relatively well known (refs. 9-17). What is not well known are its practical applications. Whether it is necessary and economical to have concurrent error detection in arithmetic processors is yet to be seen. However, at the present time, there are two principal developments in this direction — the Amdahl 470/V6 computer (ref. 18) and the Burroughs BSP computer (ref. 19).

In this paper, the method of concurrent error detection uses low-cost residue codes with a check-length ℓ and a check-base $m = 2^{\ell} - 1$ for detecting errors in arithmetic operations, and uses a single-bit parity-check code for detecting errors in logical operations. The presentation is divided into four sections. Section III describes binary number representations and shows that either the signed-magnitude or the 1's-complement, but not the 2's-complement, is more suitable for residue codes. This is because both the signed-magnitude and the 1's-complement representations have a numerical range of $M = 2^n - 1$. If ℓ divides n, then m divides M, so that $(2^n - 1) \bmod m = 0$, and this greatly simplifies the check equations. Section IV describes two methods for residue generation. Section V describes residue codes for checking arithmetic operations of $+$, $-$, $\times$, $\div$, complement, shift, and rotate. Finally, section VI describes a single-bit parity-check code for checking logical operations of XOR, OR, and AND, and also arithmetic operations of complement, shift, and rotate.

The author wishes to thank K. G. Stevens, Jr. of the NASF Project and D. K. Stevenson of the Institute for Advanced Computation for their reading and commenting on the work reported herein.

III. NUMBER REPRESENTATIONS AND RESIDUE CODES

The hardware implementation complexity of residue codes is directly related to the choice of number representation and also to the check-base. In this section it is shown that either the signed-magnitude or the 1's-complement, but not the 2's-complement, is more suitable for residue codes. Furthermore, if the check-base m is selected to be $m = 2^{\ell} - 1$, then the code is a low-cost residue code (ref. 10). It is low-cost in the sense that the residue of a binary number N modulo m can be obtained by additions (without actual divisions).

In present practices, there are three methods for representing binary numbers in a computer: the signed-magnitude, 1's-complement, and 2's-complement representations. Let N be an n-bit binary number consisting of a sign bit and $n - 1$ magnitude bits. Without losing generality, N is treated in this paper as an integer (not a fraction) as

$$N = b_s b_{n-2} b_{n-3} \cdots b_1 b_0 \tag{1}$$

or N can be written in its natural value form as

$$N = \sum_{i=0}^{n-1} b_i 2^i \tag{2}$$

where the sign bit of this equation requires a different interpretation for each of the three number representations.

Let M be the numerical range of N, that is, the range of numbers that can be represented by the n bits of N. For both the signed-magnitude and

the 1's-complement representations, $M = 2^n - 1$, and the range of numbers are $-(2^{n-1} - 1), -(2^{n-1} - 2), \ldots, -1, 0, +1, +2, \ldots, +(2^{n-1} - 2), +(2^{n-1} - 1)$. For the 2's-complement representation, $M = 2^n$, and the numbers range from -2^{n-1} to $+(2^{n-1} - 1)$. These two ranges, M, are primarily used in number complementation and in subtraction. As is well known, subtraction in the 1's-complement or the 2's-complement representation is a trivial operation. In the signed-magnitude representation, subtraction is simply a complementation followed by an addition.

Let m be the modulus on the congruence of N, or the check-base in the residue computation of N. The residue of a negative number in complement form can be computed by noting that if a positive number has residue r, the negative number will have residue $-r$. Because of the convention that the check bits of N should represent the least positive residue, $-r$ should be converted to a positive residue as

$$(m - r) \equiv (-r) \bmod m \tag{3}$$

If N is negative, its complemented form is either

$$2^n - 1 - N \quad \text{(1's-complement)} \tag{4}$$

or

$$2^n - N \quad \text{(2's-complement)} \tag{5}$$

The residue of these will be, respectively,

$$(2^n - 1 - N) \bmod m \equiv (2^n - 1) \bmod m + m - (N) \bmod m \tag{6}$$

$$(2^n - N) \bmod m \equiv (2^n) \bmod m + m - (N) \bmod m \tag{7}$$

Equations (6) and (7) indicate that if a number is in the complemented form and the residue is calculated using the residue generator for positive numbers, the following constant must be subtracted from the result to get the true residue.

$$(2^n - 1) \bmod m \quad \text{(1's-complement)}$$

$$(2^n) \bmod m \quad \text{(2's-complement)}$$

If m is selected to be $(2^\ell - 1)$, then it is known from number theory that

$$(2^n - 1) \bmod m \equiv 0 \quad \text{if} \quad n \equiv 0 \bmod \ell \tag{8}$$

In other words, for $\ell \leq n$, $(2^\ell - 1)$ divides $(2^n - 1)$ if and only if ℓ divides n. A simple proof is as follows:

Let $m = 2^\ell - 1$. Then $2^\ell \equiv 1 \bmod m$ by the definition of congruence. If ℓ divides n, then $n = k\ell$ for some k. Also $(2^\ell)^k \equiv 1^k \bmod m$, or $2^{k\ell} = 2^n \equiv 1 \bmod m$. Hence

$$(2^n - 1) \equiv 0 \bmod (2^\ell - 1),$$

which means $(2^\ell - 1)$ divides $(2^n - 1)$ by the definition of congruence.

From equation (8), it is clear that the 1's-complement, and hence the signed-magnitude, representation clearly has an advantage over the 2's-complement representation in residue checking. In this case, the additive constant to make the residue of a negative number the same as for a positive number is 0 for 1's-complement (also signed-magnitude). Some values of m for which this works are as follows:

ℓ	m	n
2	3	2, 4, 6, ---, 24, 36, 48, 72, ---
3	7	3, 6, 9, ---, 24, 36, 48, 72, ---
4	15	4, 8, 12, ---, 24, 36, 48, 72, ---

Another advantage of selecting $m = 2^\ell - 1$ is the ease of residue generation in that the residue can be obtained using only additions, not divisions, as described in the next section.

IV. RESIDUE GENERATION

The residue generator is the principal piece of additional hardware required to allow the use of residue codes for concurrent error detection in the arithmetic processor. For this reason, the check-base m must be selected so that the residue generator can be implemented for simplicity, high-speed operation, and self-checking if possible. In this section, two methods of residue generation are described. Both methods chose $m = 2^\ell - 1$ for low-cost residue codes and high-speed operation. One method is not self-checking while the other method is capable of self-checking against a single failure within the generator.

Let $m = 2^\ell - 1$ be the check-base. For this value of m, the residue can be represented by ℓ bits. Let the n-bit integer N be partitioned into t bytes of length ℓ where $(t - 1)\ell < n \leq t\ell$. Let these bytes be $B_{t-1}B_{t-2} \cdots B_1B_0$. Then N from equation (2) can be written as

$$N = \sum_{i=0}^{t-1} B_i 2^{\ell i} \quad \text{where} \quad 0 \leq B_i < 2^\ell \tag{9}$$

Since $2^\ell \equiv 1 \bmod (2^\ell - 1)$, it follows that

$$N \bmod (2^\ell - 1) \equiv \sum_{i=0}^{t-1} B_i (2^\ell)^i \bmod (2^\ell - 1)$$

$$\equiv \sum_{i=0}^{t-1} B_i \bmod (2^\ell - 1) \tag{10}$$

Thus N modulo $(2^{\ell} - 1)$ can be computed by additions only (without divisions). That is, it can be computed by modulo$(2^{\ell} - 1)$ summation of t ℓ-bit bytes of N.

Addition modulo$(2^{\ell} - 1)$ can be performed by using an ordinary binary adder with end-around carry, or by logic decoding as suggested by Pertman (ref. 13). Such codes generated by $m = 2^{\ell} - 1$ are called low-cost residue codes by Avizienis (ref. 10). An example of an arithmetic processor using such a code with $\ell = 2$ and $m = 3$ reported in open literature is the Burroughs BSP arithmetic element (ref. 19). The design of the modulo-3 generator integrated-circuit chip for the BSP is also reported in open literature (ref. 20).

A slightly different method for residue generation was described by Kolupaev (ref. 21) in designing self-testing residue trees. This method differs from the method of equation (10) in that in the first level of computation, the width of the byte is not restricted to ℓ bits. Let $|N|_m$ denote the residue of N modulo m. Because of the homomorphism relating modulo m addition with ordinary addition, the residue of N modulo m in equation (9) for a k-bit byte width, usually $k \geq \ell$, is

$$|N|_m = \left|B_{t-1}2^{k(t-1)}\right|_m + \left|B_{t-2}2^{k(t-2)}\right|_m + \ldots + |B_1 2^k|_m + |B_0|_m \tag{11}$$

Furthermore, if k is chosen so that the residues of 2^{ki}, $i = 0, 1, 2, \ldots, t-1$ are 1 modulo m, then equation (11) can be reduced to

$$|N|_m = |B_{t-1}|_m + |B_{t-2}|_m + \ldots + |B_1|_m + |B_0|_m \tag{12}$$

One such combination of numbers is $m = 3$ and $k = 4$ so that 2^{ki}, $i = 0, 1, 2, \ldots$, are $1, 2^4, 2^8, 2^{12}, \ldots$. The relationship of $2^{ki} \equiv 1 \bmod m$ simply implies that ℓ divides k. This again leads to the fact that if ℓ divides k, then ℓ divides ki. Therefore, $(2^{\ell} - 1)$ divides $(2^{ki} - 1)$ so that $2^{ki} \equiv 1 \bmod m$.

A comparison of residue generation using equations (10) and (12) are shown in figures 1 and 2, respectively. It should be noted that the residue generator in figure 2 is self-checking as described by Kolupaev (ref. 21).

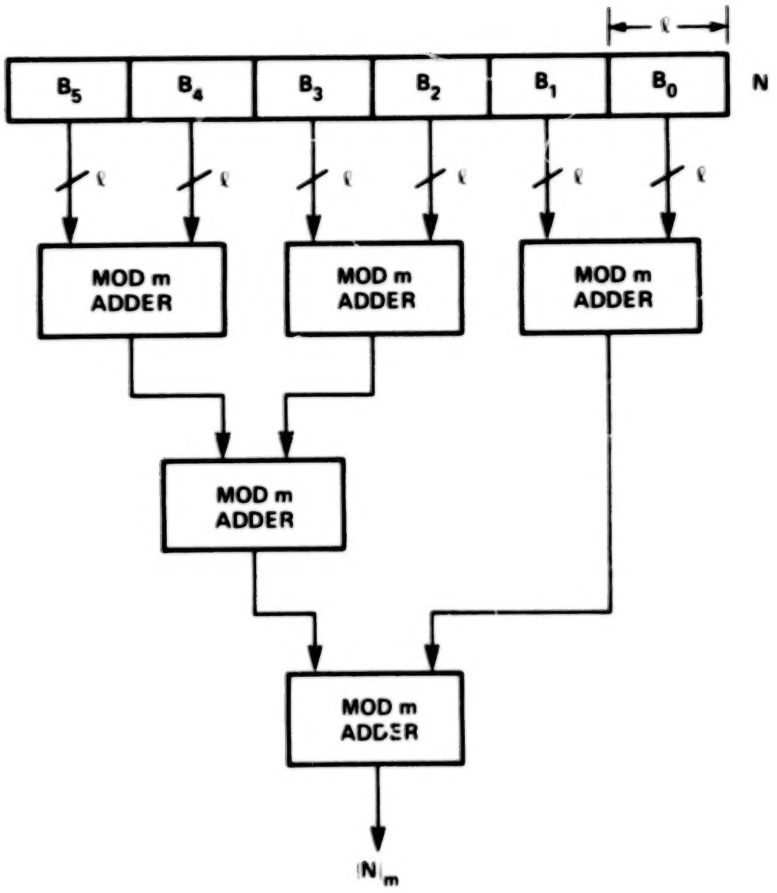


Figure 1.- Residue generation using modulo m adders with $m = 2^{\ell} - 1$.

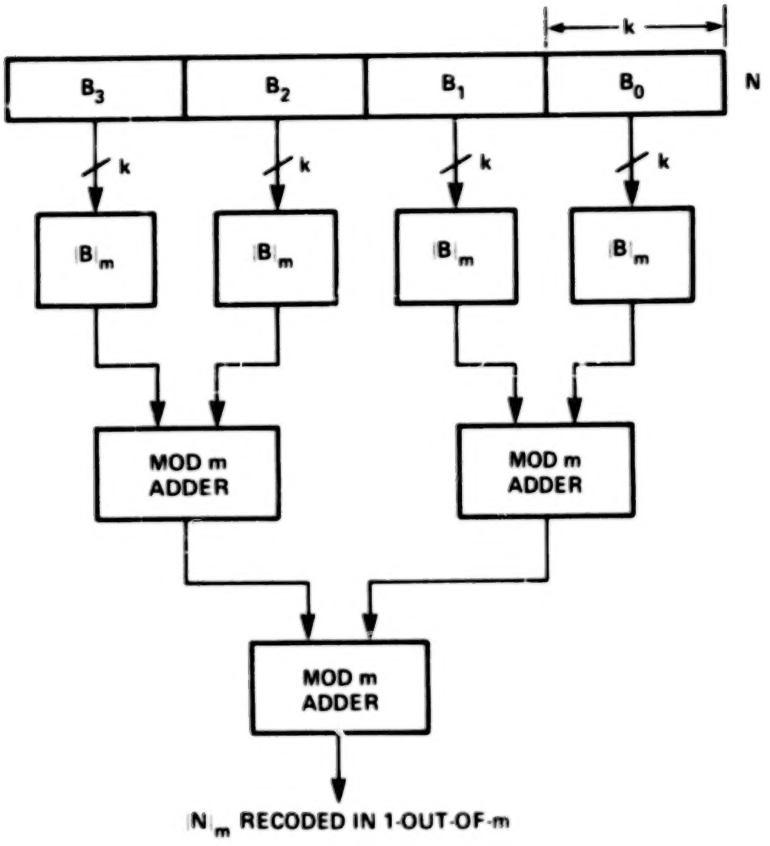


Figure 2.- Self-testing residue generator (Kolupaev), $k \geq \ell$.

V. ARITHMETIC ERROR DETECTION BY RESIDUE CODES

In this section, residue codes for concurrent arithmetic error detection are described for addition, subtraction, multiplication, division, complement, shift, and rotate. Because of the complexity of presenting general residue checking equations for the general check-base m that is valid for all three number representations, the description is restricted (1) to $\ell = 2$ and $m = 3$, and (2) to only the signed-magnitude and the 1's-complement representations. With this restriction, the description presented here can still be extended to other choices of m and also to the 2's-complement representation. As described under Number Representations, the number N has n bits, with n even or odd, and the range is $M = 2^n - 1$.

It should be pointed out that for the signed-magnitude representation, the sign bit is not a value bit and, therefore, it does not enter the arithmetic computation along with the magnitude bits. For this reason and for reason of notational convenience, n in the range $M = 2^n - 1$ should be interpreted as all magnitude bits, not sign plus $n - 1$ bits of magnitude as in the case for the 1's-complement representation.

In the following discussion, all additions and multiplications are modulo 3. For a negative number A, its residue generated by a positive number mod-3 generator is $(3 - |A|_3)$ automatically; hence, the term $(3 - |A|_3)$ cannot be separated into $(3) - (|A|_3)$.

For $M = 2^n - 1$, it can be shown that $|M|_3 = 0$ for n even and $|M|_3 = 1$ for n odd. If n is even, then 2 divides n and hence $m = 3$ divides M. Therefore, $|M|_3 = 0$ for n even. If n is odd, then write n as $n_e + 1$, where n_e is an even number. The congruence

$$|2^{n_e+1} - 1|_3 = |2^{n_e}|_3 \times |2|_3 - |1|_3 = 1$$

From these discussions, it follows that $|2^c|_3 = 1$ for c even and $|2^c|_3 = 2$ for c odd. Also, in modulo 3 residue calculation, $|2x|A||_3$ is equivalent to exchanging the positions of the two bits of $|A|_3$, which is equivalent to the complement of $|A|_3$, that is, $|2x|A||_3 = \overline{|A|_3}$.

As will be described later, the check equations for n odd is more complex than for n even. For this reason, it might be advantageous in practice to pad an extra bit in order to make n even.

Addition and Subtraction

In binary arithmetic, addition and subtraction are equivalent. In subtraction, the subtrahend is usually 1's-complemented and then proceeds with the addition operation. Therefore, only residue checking for addition is described. In the open literature, an excellent algorithm for addition and subtraction used in a real computer can be found in a paper by Davis (ref. 22). The description presented in the sequel is restricted to fixed-point addition

since the other operational steps of floating-point addition such as shift and compare can be checked separately. Also, it is assumed that overflow conditions will not occur here, or that they can be detected separately using standard techniques.

Let A and B, $0 \leq A, B < M$ be two n-bit numbers to be added. There are four cases to be considered.

CASE 1. A = +, B = +, and SUM = (A + B) = +

The check equation is

$$|SUM|_3 = |A + B|_3 = |A|_3 + |B|_3 \tag{13}$$

CASE 2. A = +, B = -, $|A| > |B|$, and SUM = +

The check equation is

$$\begin{aligned} |SUM|_3 &= |A + (M - B)|_3 \\ &= |A|_3 + |M|_3 + (3 - |B|_3) \end{aligned} \tag{14}$$

where $|M|_3 = 0$ for n even, 1 for n odd.

CASE 3. A = +, B = -, $|A| < |B|$, and SUM = -

Since the SUM is negative, it is of the form

$$SUM = M - (A + B)$$

and this must be equal to

$$A + (M - B)$$

Taking residue modulo 3 of these two equations, the result is

$$|M|_3 + (3 - |A + B|_3) = |A|_3 + |M|_3 + (3 - |B|_3)$$

which can be reduced to

$$(3 - |A + B|_3) = |A|_3 + (3 - |B|_3)$$

Therefore, the check equation is

$$|SUM|_3 = |A|_3 + (3 - |B|_3) \tag{15}$$

CASE 4. A = -, B = -, and SUM = -

Since the SUM is negative, it is of the form

$$SUM = M - (A + B)$$

and this must be equal to

$$(M - A) + (M - B)$$

Therefore, the check equation is

$$|SUM|_3 = |M|_3 + (3 - |A|_3) + (3 - |B|_3) \tag{16}$$

where $|M|_3 = 0$ for n even, 1 for n odd.

An analysis of the above four cases indicated that a correction factor of 1 is required if n is odd for cases 2 and 4. In practice, it is probably more economical to pad an extra bit to make n even than to account for the correction.

Multiplication

In the NASF system, only very high-speed multiplication algorithms are of interest. One such algorithm is the use of mxm bits multiplier chips followed by one row of (p,2) counters (adders) to compress $p = 1 + 2(\ell - 1)$ summands all at once without carry propagations (ref. 23). If one of these high-speed algorithms is used, then the entire multiplication unit can be checked by residue codes. In the following, only fixed-point multiplication is described. For floating-point multiplication, the normalization and the rounding operations must also be checked.

The residue checking of signed-magnitude multiplication is very simple because the sign bit can be separated from the magnitude bits, and hence, all multiplications can be treated as positive numbers. The check equation for the product, PROD, is simply

$$|PROD|_3 = |A|_3 \times |B|_3 \tag{17}$$

The residue checking of 1's-complement multiplication is more complex. There are four cases to be considered.

CASE 1. A = +, B = +, and PROD = +

The check equation is

$$|PROD|_3 = |A|_3 \times |B|_3 \tag{18}$$

CASE 2. A = +, B = -, and PROD = -

Since the product is negative, it is of the form

$$M - (A \times B)$$

and it must be equal to

$$(A) \times (M - B)$$

Taking residue modulo 3 of the two equations, the result is

$$|M|_3 + (3 - |A \times B|_3) = |A|_3 \times [|M|_3 + (3 - |B|_3)]$$

If $|M|_3 = 0$ for n even, the check equation is

$$|PROD|_3 = (3 - |A \times B|_3)$$

$$= |A|_3 \times (3 - |B|_3) \qquad (19)$$

If $|M|_3 = 1$ for n odd, then

$$1 + (3 - |A \times B|_3) = |A|_3 + |A|_3 \times (3 - |B|_3)$$

and the check equation is

$$|PROD|_3 = (3 - |A \times B|_3)$$

$$= |A|_3 \times (3 - |B|_3) + CF \qquad (20)$$

where

$$CF = |A|_3 - 1$$

CASE 3. A = -, B = +, and PROD = -

This case is similar to case 2, and the check equations are

$$|PROD|_3 = (3 - |A|_3) \times |B|_3 \quad \text{for} \quad |M|_3 = 0 \qquad (21)$$

$$|PROD|_3 = (3 - |A|_3) \times |B|_3 + CF \quad \text{for} \quad |M|_3 = 1 \qquad (22)$$

where

$$CF = |B|_3 - 1$$

CASE 4. A = -, B = -, and PROD = +

Since the product is positive, its equation is

$$PROD = A \times B = (M - A) \times (M - B)$$

It follows that the check equations are

$$|PROD|_3 = (3 - |A|_3) \times (3 - |B|_3) \quad \text{for} \quad |M|_3 = 0 \tag{23}$$

$$|PROD|_3 = (3 - |A|_3) \times (3 - |B|_3) + CF \quad \text{for} \quad |M|_3 = 1 \tag{24}$$

where

$$CF = 1 + (3 - |A|_3) + (3 - |B|)$$

An analysis of the above four cases indicates that a correction factor, CF, is not required if $|M|_3 = 0$ for n even. For n odd, $|M|_3 = 1$, and a correction factor is required whenever one or both operands are negative.

Division

Currently, there are two types of division algorithms commonly used. One type gives only the quotient and no remainder, and uses multiplication for iterative convergence. In this case, the division is checked automatically if the multiplication is checked. The other type gives both the quotient and the remainder, and these algorithms generally use the shift-and-subtract method. Again, if the shift and add logics are checked, then the division is also checked automatically.

In the above discussion, the division may be slowed up if each iterative step is checked. As an alternative, the division of A/B can be checked according to

$$A = BQ + r$$

Where Q is the quotient and r is the remainder, the check equation is

$$|A|_3 - |r|_3 = |B|_3 \times |Q|_3 \tag{25}$$

Complement

Let A be an n-bit vector, and let the complement (1's-complement) of A be denoted $\bar{A}$. Then the check equation is

$$|\bar{A}|_3 = |M - A|_3$$

$$= (3 - |A|_3) \quad \text{if} \quad |M|_3 = 0, \text{ n even} \tag{26}$$

$$= 1 + (3 - |A|_3) \quad \text{if} \quad |M|_3 = 1, \text{ n odd} \tag{27}$$

The above equations indicate that the complement of A can be checked by: (1) calculate $|A|_3$ from $\bar{A}$, and then obtain $(3 - |A|_3)$ by some combinational technique, (2) calculate $|\bar{A}|_3$, and (3) compare $|\bar{A}|_3$ with $(3 - |A|_3)$ for n even and $(3 - |A|_3) + 1$ for n odd.

End-off Shift

The end-off shift operation consists of moving a number to the left (or right) a specified number of places. Depending upon the operation being performed, the bit positions vacated at the right (or left) end of the shifted number might be filled with either 0's or 1's. In the following discussions, let A be the original number and A_c be the number A left (or right) shifted by c bits, $c \leq n$.

For the left shift, A and A_c are as shown in figure 3(a). The value of A_c can be written as

$$A_c = 2^c Z + W$$

where W is the c-bit string shifted in from the right end. There are two cases: (1) $W = 00 \ldots 0$, and (2) $W = 11 \ldots 1$.

CASE 1. $W = 00 \ldots 0$

The check equations are

$$|A_c|_3 = |2^c Z|_3 = |2^c|_3 \times |Z|_3$$

$$= |Z|_3 \qquad \text{for } c \text{ even} \tag{28}$$

$$= 2 \times |Z|_3 \qquad \text{for } c \text{ odd} \tag{29}$$

The check process of the above equations can be summarized as follows:

a. Obtain Z from A by resetting the Y part of A to zero. Calculate $|Z|_3$ and $|A_c|_3$.

b. Check $|A_c|_3 = |Z|_3$ for c even, or $|A_c|_3 = 2|Z|_3$ for c odd.

CASE 2. $W = 11 \ldots 1$

The check equations are

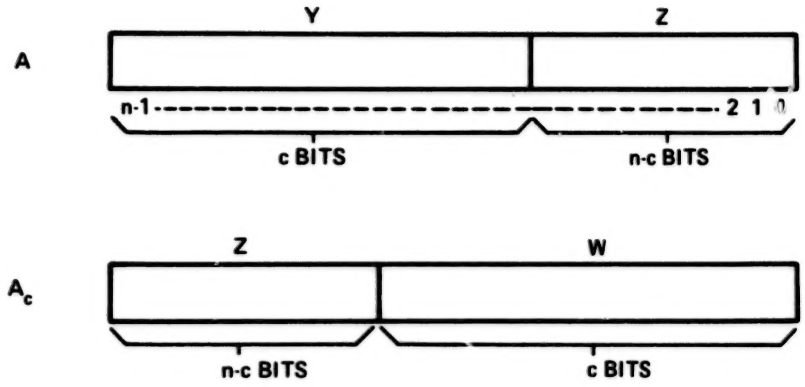
$$|A_c|_3 = |2^c Z| + |W|_3$$

$$= |2^c|_3 \times |Z|_3 + |2^c - 1|_3$$

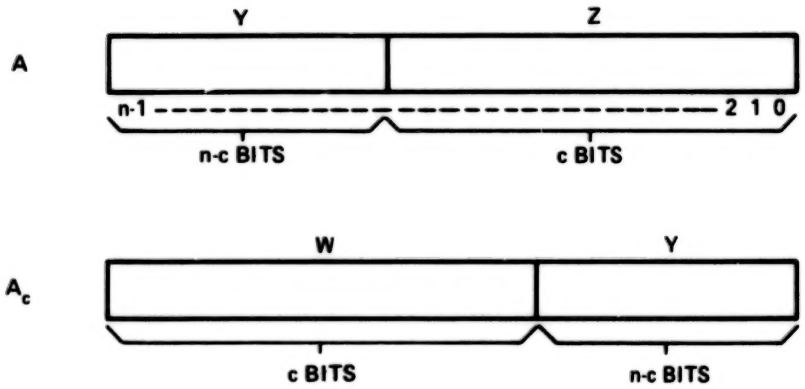
$$= |Z|_3 \qquad \text{for } c \text{ even} \tag{30}$$

$$= 2 \times |Z|_3 + 1 \qquad \text{for } c \text{ odd} \tag{31}$$

and the check process is similar to case 1.



(a) LEFT SHIFT END-OFF BY c BITS



(b) RIGHT SHIFT END-OFF BY c BITS

Figure 3.- Shift operations, W = 00 . . . 0 or 11 . . . 1.

For the right shift, A and A_c are as shown in figure 3(b). The value of A_c can be written as

$$A_c = 2^{n-c}W + Y$$

where W is the c-bit string shifted in from the left end. Again, there are two cases: W = 00 . . . 0 and W = 11 . . . 1.

CASE 1. W = 00 . . . 0

The check equation is simply

$$|A_c|_3 = |Y|_3 \tag{32}$$

and the check process is as follows:

a. Obtain 2^cY from A by resetting the right most c bits. Calculate $|2^cY|_3$ and $|A_c|_3$. The value $|2^cY|_3$ is $|Y|_3$ for c even and $2|Y|_3$ for c odd.

b. Check $|A_c|_3 = |2^cY|_3 = |Y|_3$ for c even. For c odd, check $2|A_c|_3 = \overline{|A_c|}_3 = 2|Y|_3 = |2^cY|_3$.

CASE 2. W = 11 . . . 1

From the equation $A_c = 2^{n-c}W + Y$, the $(2^{n-c}W)$ part has value $2^n - 2^{n-c}$. Thus

$$A_c = (2^n - 2^{n-c}) + Y$$

The check equations are

$$|A_c|_3 = |2^n - 2^{n-c}|_3 + |Y|_3$$

$$= |Y|_3 \qquad \text{for n even or odd, and c even} \tag{33}$$

$$= |Y|_3 - 1 \qquad \text{for n even and c odd} \tag{34}$$

$$= |Y|_3 + 1 \qquad \text{for n odd and c odd} \tag{35}$$

The check process is similar to case 1 above except that there are three comparisons to be made depending upon the evenness and oddness of n and c.

Rotate

Residue checking for left or right rotate operations is simple if $|M|_3 = 0$ for n even, and is difficult if $|M|_3 = 1$ for n odd. The

mathematical developments for these two cases are different, and the case for n even is derived first. Refer to figure 4 for the following discussions:

CASE 1. $|M|_3 = 0$ for n even

Let A_c be the result of rotating A left by c bits for $0 \leq c \leq n$. The result of A_c modulo M is equivalent to multiplying A by 2^c modulo M, and can be shown as follows:

From figure 4(a),

$$|A_c|_M = |Y|_M + |2^c Z|_M$$

and

$$|2^c A|_M = |2^c \times (Z + 2^{n-c} Y)|_M$$

$$= |2^c Z|_M + |2^n Y|_M$$

$$= |Y|_M + |2^c Z|_M$$

Thus, $|A_c|_M = |2^c A|_M$ and this concludes the proof. The above proof immediately leads to

$$|A_c|_3 = \left| |2^c A|_M \right|_3$$

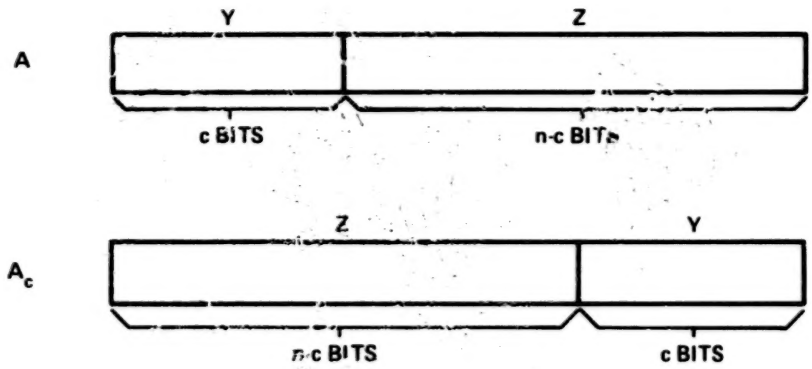
Since $\left| |2^c A|_M \right|_3 = |2^c A|_3$ if $|M|_3 = 0$, then the check equations are

$$|A_c|_3 = |2^c A|_3$$

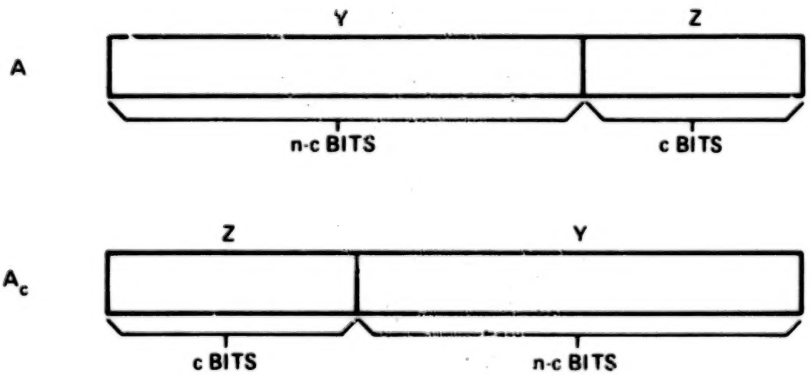
$$= |A|_3 \qquad \text{for c even} \qquad (36)$$

$$= 2|A|_3 \qquad \text{for c odd} \qquad (37)$$

Right rotate by c bits is equivalent to left rotate by n - c bits. Since n is even as stated, then n - c is odd if c is odd. Therefore, the check equations for right rotation are exactly the same as the check equations for left rotation.



(a) LEFT ROTATION BY c BITS



(b) RIGHT ROTATION BY c BITS

Figure 4.- Rotate operations.

CASE 2. $|M|_3 = 1$ for n odd

For left rotation

$$|A_c|_3 = |2^c Z + Y|_3$$

$$= |Z|_3 + |Y|_3 \quad \text{for c even} \tag{38}$$

$$= 2|Z|_3 + |Y|_3 \quad \text{for c odd} \tag{39}$$

For right rotation

$$|A_c|_3 = |2^{n-c} Z + Y|$$

$$= 2|Z|_3 + |Y|_3 \quad \text{for c even} \tag{40}$$

$$= |Z|_3 + |Y|_3 \quad \text{for c odd} \tag{41}$$

VI. LOGICAL ERROR DETECTION BY PARITY CHECKING

In this section, a single-bit parity-check code for concurrent error detection is described for complement, shift, rotate, XOR, OR, and AND. Parity checks, when compared to residue checks, are very simple for these operations. In the following discussions, A and B are n-bit vectors, where n can be even or odd.

Complement

The parity modulo 2 of the complement of A is very easy to predict. Let $\bar{A}$ be the complement of A. If A has parity P(A), where P(A) = 0 or 1, then the parity of $\bar{A}$ is

$$P(\bar{A}) = P(A) \quad \text{for n even} \tag{42}$$

$$= \overline{P(A)} \quad \text{for n odd} \tag{43}$$

End-off Shift

For all end-off shift operations, refer to figure 3 for reference. Let P(W) be the parity of W, P(Y) be the parity of Y, and P(Z) be the parity of Z. The parity of A is

$$P(A) = P(Y) \oplus P(Z)$$

For left shift, the parity of A_c, and hence the check equation, is

$$P(A_c) = P(Z) \oplus P(W) = P(W) \oplus P(A) \oplus P(Y) \tag{44}$$

For right shift, the parity of A_c, and hence the check equation is

$$P(A_c) = P(W) \oplus P(Y) = P(W) \oplus P(A) \oplus P(Z) \tag{45}$$

It should be noted that both $P(Y)$ and $P(Z)$ are the parity bits of the shifted out bit strings.

Rotate

For all rotate operations, refer to figure 4 for reference. Since a rotate operation merely rotates c bits around, there is no gain or loss in the total number of 0's or 1's. Therefore, the check equation is

$$P(A_c) = P(A) \tag{46}$$

for all cases.

XOR

The bit-by-bit XOR operation is closed for the single-bit parity-check code; that is, the XOR of any two code words is also a code word. Let A have parity $P(A)$, B have parity $P(B)$, and $XOR = A \oplus B$ have parity $P(XOR)$. Then

$$P(A) = a_{n-1} \oplus a_{n-2} \oplus \ldots \oplus a_1 \oplus a_0$$

$$P(B) = b_{n-1} \oplus b_{n-2} \oplus \ldots \oplus b_1 \oplus b_0$$

and the check equation is

$$\begin{aligned} P(XOR) &= P(A \oplus B) \\ &= (a_{n-1} \oplus b_{n-1}) \oplus \ldots \oplus (a_1 \oplus b_1) \oplus (a_0 \oplus b_0) \\ &= (a_{n-1} \oplus a_{n-2} \oplus \ldots \oplus a_1 \oplus a_0) \\ &\quad \oplus (b_{n-1} \oplus b_{n-2} \oplus \ldots \oplus b_1 \oplus b_0) \\ &= P(A) \oplus P(B) \end{aligned} \tag{47}$$

OR and AND

The bit-by-bit OR and AND operations are not closed under the parity-check codes. Although these two operations can be checked by residue codes as described by Monteiro and Rao (ref. 15), they are probably best checked by the parity method suggested by Sellers et al. (ref. 12). In their method, they suggested an augmented adder to check the OR operation by noting that

$$(A \oplus B) \oplus (AB) = A + B$$

and to check the AND operation by duplication.

In checking the OR operation, the SUM and CARRY outputs of the augmented adder are

$$SUM = (A \oplus B) \oplus (AB) = A + B$$

$$CARRY = AB$$

Let P(SUM) be the parity of the SUM output, and P(CARRY) be the parity of the CARRY output. Then the check equation for the OR operation is

$$P(SUM) = P(A) \oplus P(B) \oplus P(CARRY) \tag{48}$$

In checking the AND operation, both the SUM and the CARRY outputs are

$$SUM = CARRY = AB$$

Thus, the AND operation can be checked by duplication, and the check equation is

$$P(SUM) = P(CARRY)$$

Another method for checking OR and AND (suggested by D. Stevenson during the review of this paper by noting the SUM equation above) is by the use of the XOR property according to the identity of

$$A \oplus B = (A + B) \oplus (AB)$$

From this equation, it follows that

$$P(A \oplus B) = P(A + B) \oplus P(AB)$$

$$= P(A) \oplus P(B)$$

Therefore, OR and AND can be checked as follows:

$$P(A + B) = P(A) \oplus P(B) \oplus P(AB)$$

$$P(AB) = P(A) \oplus P(B) \oplus P(A + B)$$

Depending upon the logic implementation, it appears that this method suggested by Stevenson may be simpler than the method suggested by Sellers et al. (ref. 12).

VII. CONCLUSION

The culmination of an effort to develop a low-cost method for concurrent error detection for arithmetic processors, large and small, has been presented in this paper. The method uses a low-cost separate residue code to check arithmetic operations, and uses a single-bit parity-check code to check logical operations. Assume the processor to be checked has a word length of n bits, the method shows that:

1. For checking arithmetic operations of +, -, $\times$, and $\div$, a low-cost residue code with check-base $m = 2^{\ell} - 1$, $\ell \leq n$, ℓ divides n, and n even, is simple and economical.

2. For checking arithmetic operations of complement, shift, and rotate, either a residue code or a single-bit parity-check code can be used. Both methods are simple and economical. In terms of logic implementation complexity, the parity-check code is simpler than the residue code.

3. For checking logical operations of XOR, OR, and AND, the single-bit parity-check code is the simplest method to use and to implement.

4. For using a residue code to check arithmetic operations, the signed-magnitude or the 1's-complement representation, not the 2's-complement representation, should be used. This is because these two representations have a numerical range of $M = 2^n - 1$. If ℓ divides n, them m divides M, so that $(2^n - 1) \bmod m = 0$, and this simplifies greatly the check equations. In residue checking of multiplication and division, the signed-magnitude representation is the simplest to use.

5. For using the single-bit parity-check code to check XOR, OR, and AND in conjunction with an augmented adder, this adder design is considerably different from the well-known STTL 74S181, which is currently a standard commercial arithmetic-logic unit.

From the above five points, it appears that the architecture of the processor is best structured with the following functional units:

1. A floating-point add unit

2. A multiply/divide unit

3. An integer unit capable of performing integer arithmetics and logical operations

This paper did not proceed into the logic design and implementation of the suggested method of concurrent error detection. Such an exercise would

certainly be very laborious and also beyond the scope of this paper. However, based upon practical experiences and today's integrated-circuit technology, it can be conjectured that the added redundancy for checking should not exceed 20% of the processor complexity, and that the method should provide a self-checking coverage of at least 80% of the processor.

The problem of residue generation is the basic obstacle, and hence the cost, of using residue codes for self-checking. For this problem, the best current solution is probably an LSI implementation of the self-testing residue generator suggested by Kolupaev (ref. 21).

Although the results given in this paper are very encouraging, the problem of applying them to the design of a processor still requires a large amount of effort. In any case, the search for methods of concurrent error detection for processors is rapidly converging and considerably narrowed.

Finally, the problem of designing a self-checking computer system should take a top-down approach and consider the problem from the overall system viewpoint, at least to include the memory and the processor. At present, the memory is protected by a modified Hamming code and the processor is self-checked by a combination of residue codes and parity-check codes. This is a mismatch! What is needed is one uniform coding system, perhaps a biresidue code or some modified linear residue codes, that can protect both the memory and the processor. At present, however, it is not known whether or not such a coding system can be found to give a cost effective performance.

Ames Research Center
National Aeronautics and Space Administration
Moffett Field, California 94035, March 21, 1979

REFERENCES

1. Preliminary Study For a Computational Aerodynamic Design Facility. NASA, Proposal 2-26539 (AS), October 1976.

2. Numerical Aerodynamic Simulation Facility, Preliminary Study. Final Report, NASA CR-152061 and CR-152062, October 1977, Burroughs Corporation, Paoli, Pennsylvania, Contract NAS2-9456.

3. Numerical Aerodynamic Simulation Facility, Preliminary Study Extension. Final Report, NASA CR-152107, February 1978, Burroughs Corporation, Paoli, Pennsylvania, Contract NAS2-9456.

4. Numerical Aerodynamic Simulation Facility, Preliminary Study. Final Report, NASA CR-152059, October 1977, Control Data Corporation, Minneapolis, Minnesota, Contract NAS2-9457.

5. Numerical Aerodynamic Simulation Facility, Preliminary Study Extension. Final Report, N78-19783, February 1978, Control Data Corporation, Minneapolis, Minnesota, Contract NAS2-9457.

6. Feierbach, G.; and Stevenson, D. K.: The Phoenix Array Processing System. Parts I and II. Phoenix Project Memo No. 007, Institute for Advanced Computation, Sunnyvale, California, November 1978.

7. Avižienis, A.: Fault-Tolerance Considerations for Large-Scale, High-Speed Numerical Computers. Final Report for Contract NASA-Ames Research Center, December 1978, Contract A49214B.

8. Lim, R. S.: Fault-Tolerant Computing: A Preamble for Assuring Viability of Large Computer Systems. NASA TP-1067, October 1977.

9. Avižienis, A.: Concurrent Diagnosis of Arithmetic Processors. Digest of First Annual IEEE Computer Conference, September 1967, pp. 34-37.

10. Avižienis, A.: Digital Fault Diagnosis by Low-Cost Arithmetical Coding Techniques. Proceedings of the Purdue Centennial Year Symposium Information Processing, vol. 1, Purdue Univ. Eng. Exp. Sta., Lafayette, Indiana, April 1969, pp. 81-91.

11. Avižienis, A.: Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital Systems Design. IEEE Trans. Computer, vol. C-20, no. 11, November 1971, pp. 1322-1331.

12. Sellers, F. F.; Hsiao, M. Y.; Bearnson, L. W.: Error Detecting Logic for Digital Computers. McGraw-Hill Book Co., Inc., 1968.

13. Pertman, A. E.: Circuits for Checking Arithmetic Errors by Means of Residue Coding. Report No. NOLTR 69-38, U.S. Naval Ordnance Lab., Silver Springs, Maryland, February 1969.

14. Rao, T. R. N.: Error-Checking Logic for Arithmetic-Type Operations of a Processor. IEEE Trans. Computer, vol. C-17, no. 9, September 1968, pp. 845-849.

15. Monteiro, P.; and Rao, T. R. N.: A Residue Checker for Arithmetic and Logical Operations. Digest of Papers from the 1972 IEEE Proceedings of International Symposium on Fault-Tolerant Computing, Newton, Massachusetts, June 19-21, 1972, pp. 8-13.

16. Rao, T. R. N.: Error Coding for Arithmetic Processors. Academic Press, 1974.

17. Wakerly, J. F.: Error Detecting Codes, Self-Checking Circuits and Applications. North-Holland, 1978.

18. Amdahl 470 V/6 Features. Amdahl Corporation, Sunnyvale, California, 1975.

19. Gajski, D. D.; and Rubinfield, L. P.: Design of Arithmetic Elements for Burroughs Scientific Processor. Proceedings of the Fourth IEEE Symposium on Computer Arithmetic, Santa Monica, California, October 1978, pp. 245-256.

20. Gajski, D. D.; and Vora, C.: High-Speed Modulo-3 Generator. Electron Letters, vol. 13, no. 25, December 1977, pp. 770-772.

21. Kolupaev, S. G.: Self-Testing Residue Trees. TR-49, Digital Systems Lab., Stanford University, August 1978.

22. Davis, R. L.: The ILLIAC IV Processing Element. IEEE Trans. Computers, vol. C-18, no. 9, September 1969, pp. 800-816.

23. Lim, R. S.: High-Speed Multiplication and Multiple Summand Addition. Proceedings of the Fourth IEEE Symposium on Computer Arithmetic, Santa Monica, California, October 1978, pp. 149-153.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TP-1528		

4. Title and Subtitle	5. Report Date
CONCURRENT ERROR-DETECTING CODES FOR ARITHMETIC PROCESSORS	August 1979
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Raymond S. Lim	A-7810
	10. Work Unit No.
	366-18-50-00-00

9. Performing Organization Name and Address	11. Contract or Grant No.
Ames Research Center, NASA Moffett Field, Calif. 94035	
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Paper
National Aeronautics and Space Administration Washington, D.C. 20546	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This paper describes a method of concurrent error detection for arithmetic processors. Low-cost residue codes with check-length ℓ and check-base $m = 2^{\ell} - 1$ are described for checking arithmetic operations of +, −, ×, ÷, complement, shift, and rotate. Of the three number representations, the signed-magnitude representation is preferred for residue checking. Two methods of residue generation are described: the standard method of using modulo m adders and the method of using a self-testing residue tree. A simple single-bit parity-check code is described for checking the logical operations of XOR, OR, and AND, and also the arithmetic operations of complement, shift, and rotate. For checking complement, shift, and rotate, the single-bit parity-check code is simpler to implement than the residue codes.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Error detection, Parity-check codes Error codes Arithmetic codes Residue codes	Unlimited STAR Category - 60

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	27	$4.00

*For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA-Langley, 1979